Python Collections

import collections

The `collections` module provides some great tools.

- namedtuple
- defaultdict
- OrderedDict
- dequeue
- Counter
- ChainMap
- UserDict, UserList, and UserString

namedtuple

A great boon for readability. Use it!

```
>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name gpa major')
>>>
>>> student = Student('Joe Smith', 3.7, 'Computer Science')
>>> student.name
'Joe Smith'
>>> student.gpa
3.7
>>> student.major
'Computer Science'
```

It's a tuple

```
>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name gpa major')
>>> student = Student('Joe Smith', 3.7, 'Computer Science')
>>> # like student.name
... student[0]
'Joe Smith'
>>> # like student.gpa
... student[1]
3.7
>>> # like student.major
... student[2]
'Computer Science'
```

It's a shortcut

```
# Compare to:
# Student = namedtuple('Student', 'name gpa major')
class Student:
    def __init__(self, name, gpa, major):
        self.name = name
        self.gpa = gpa
        self.major = major

student = Student('Joe Smith', 3.7, 'Computer Science')
```

It's Immutable

Just like a regular tuple.

Think of `nametuple` as just like a normal tuple, except you have the convenient option to refer to fields by name.

```
>>> Student = namedtuple('Student', 'name gpa major')
>>> student = Student('Joe Smith', 3.7, 'Computer Science')
>>> student.name
'Joe Smith'
>>> student.name = 'John Doe'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

2 Ways To Create It

```
>>> from collections import namedtuple
>>> # Pass in a string...
... Student1 = namedtuple('Student1', 'name gpa major')
>>>
>>> # ... or a list of strings.
... fields = ['name', 'gpa', 'major']
>>> Student2 = namedtuple('Student2', fields)
>>>
>>> # They work the same. Some prefer one or the other.
... student1 = Student1('Joe Smith', 3.7, 'Computer Science')
>>> student1.name
'Joe Smith'
>>> student2 = Student2('Joe Smith', 3.7, 'Computer Science')
>>> student2.name
'Joe Smith'
```

First Argument is Repetitive

The first argument doesn't HAVE to be the same as what you assign it to. But there is almost never a reason to make them different.

```
>>> # Confusing...
... Foo = namedtuple('Bar', 'x y z')
>>> baz = Foo(1,2,3)
>>> type(baz)
<class '__main__.Bar'>
```

Where Do You Use It?

Anywhere you use tuples, and keeping track of indexes is slightly confusing.

```
>>> t = get_book_info_tuple('978-0439064873')
>>> # author
... t[0]
'J.K. Rowling'
>>> # title
... t[1]
'Harry Potter and the Chamber of Secrets'
>>> # pub date
... t[2]
2000
>>> # page count
... t[3]
341
>>> # rating
... t[4]
4.7
```

Clearer APIs

Note also: `namedtuple` is backwards-compatible with regular tuples.

```
BookInfo = namedtuple('BookInfo', [
  'author',
  'title',
  'pub_date',
  'page_count',
  'rating',
])
```

```
>>> info = get_book_info_namedtuple('978-0439064873')
>>> info.title
'Harry Potter and the Chamber of Secrets'
>>> info[1]
'Harry Potter and the Chamber of Secrets'
>>> info.author
'J.K. Rowling'
>>> info.page_count
341
```

Convenience

Sometimes it's even more useful with simpler types. Notice the clarity here.

```
>>> # latitude & longitude
... Location = namedtuple('Location', 'latitude longitude')
>>> place = Location(37.77, -122.46)
>>> place.latitude
37.77
>>> place.longitude
-122.46
>>>
>>> # Compare to:
... place = (37.77, -122.46)
>>> place[0]
37.77
>>> place[1]
-122.46
```

defaultdict

Word counting problem:

```
>>> words = 'beauty is truth truth beauty'.split()
>>> counts = {}
>>> for word in words:
...     if word in counts:
...         counts[word] += 1
...     else:
...         counts[word] = 1
...
>>> print(counts)
{'is': 1, 'beauty': 2, 'truth': 2}
>>>
>>>
```

Notice the "if".

Code Compress

We can compress 4 lines to 1 with `defaultdict`.

```
>>> from collections import defaultdict
>>> counts = defaultdict(int)
>>> for word in words:
...     counts[word] += 1
...
>>> print(counts)
defaultdict(<class 'int'>, {'is': 1, 'beauty': 2, 'truth': 2})
```

Different Default Behavior

`defaultdict` does not raise a `KeyError`, unlike regular dictionaries.

```
>>> regular = dict()
>>> regular['apple']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'apple'
>>>
```

```
>>> from collections import defaultdict
>>> default = defaultdict(int)
>>> default['apple']
0
```

Of course, you can insert just like a regular dictionary.

```
>>> default['orange'] = 7
>>> default['orange']
7
```

The new-item-generator

The argument to `defaultdict` is a *callable*:

```
>>> my_ints = defaultdict(int)
>>> my_floats = defaultdict(float)
>>> my_strings = defaultdict(str)
>>>
>>> my_ints['foo']
0
>>> my_floats['foo']
0.0
>>> my_strings['foo']
''
```

That callable is automagically called, with no arguments, to insert the default value.

Question: How would you create a `defaultdict` that defaults to 1 (instead of 0)?

Custom Function

```
>>> def one():
...     return 1
...
>>> my_numbers = defaultdict(one)
>>>
>>> my_numbers['foo']
1
>>> my_numbers['bar'] += 1
>>> my_numbers['bar']
2
>>> my_numbers['baz'] = 5
>>> my_numbers['baz']
5
```

Example

```
>>> grams_of_metal = defaultdict(float)
>>> # Found an iron nugget
... grams_of_metal['iron'] += 97.3
>>> # Oooh! A silver piece!
... grams_of_metal['silver'] += 10.1
>>> # Another iron nugget
... grams_of_metal['iron'] += 72.8
>>> grams_of_metal['iron']
170.1
>>> grams_of_metal['silver']
10.1
>>> grams_of_metal['gold']
0.0
```

Another example

```
>>> foods = defaultdict(str)
>>> foods['Poland'] += 'Zupa,'
>>> foods['Australia'] += 'Macadamia nuts,'
>>> foods['USA'] += 'Burger,'
>>> foods['Australia'] += 'Meat pies,'
>>> foods['Poland'] += 'Pierogi,'
>>> for country, food in foods.items(): print(country + ": " + food)
...
USA: Burger,
Australia: Macadamia nuts,Meat pies,
Poland: Zupa,Pierogi,
```

Actually, for this data, what we really want is not a string, but a list or set. Let's do that!

Default collections

```
>>> foods = defaultdict(list)
>>> foods['Poland'].append('Zupa')
>>> foods['Australia'].append('Macadamia nuts')
>>> foods['USA'].append('Burger')
>>> foods['Australia'].append('Meat pies')
>>> foods['Poland'].append('Pierogi')
>>> for country, food in foods.items():
...     print(country + ": " + ", ".join(food))
...
USA: Burger
Australia: Macadamia nuts, Meat pies
Poland: Zupa, Pierogi
```

We also could have used `defaultdict(set)`.

Ordered Dictionary

Dictionaries in Python are unordered.

```
>>> # Atomic numbers of noble gasses.
... nobles = {'He': 2, 'Ne': 10,
...   'Ar': 18, 'Kr': 36, 'Xe': 54}
>>> # This effectively prints them in random order.
... for atom, number in nobles.items():
...     print('{} {}'.format(atom, number))
...
Ne 10
Kr 36
Xe 54
Ar 18
He 2
```

The OrderedDict class

```
>>> from collections import OrderedDict
>>> nobles = OrderedDict()
>>> nobles['He'] =  2
>>> nobles['Ne'] = 10
>>> nobles['Ar'] = 18
>>> nobles['Kr'] = 36
>>> nobles['Xe'] = 54
>>> # .keys(), .values() and .items() produce
... # data in the order of insertion.
... for atom, number in nobles.items():
...     print('{} {}'.format(atom, number))
...
He 2
Ne 10
Ar 18
Kr 36
Xe 54
```

Constructing OrderedDict's

You can also pass in a list (or any iterable) to the `OrderedDict` constructor.

```
>>> data = [
...    ('He', 2), ('Ne', 10), ('Ar', 18),
...    ('Kr', 36), ('Xe', 54)
...    ]
>>> nobles = OrderedDict(data)
>>>
>>> for atom, number in nobles.items():
...     print('{} {}'.format(atom, number))
...
He 2
Ne 10
Ar 18
Kr 36
Xe 54
```

OrderedDict is a dictionary

`OrderedDict` has all the same methods as regular dictionaries.

But the ordering adds some quirks.

```
>>> # Deleting and re-inserting changes the order.
... od = OrderedDict([('a', 7), ('b', 3), ('c', 11)])
>>> print(list(od.keys()))
['a', 'b', 'c']
>>> del od['b']
>>> od['b'] = 7
>>> print(list(od.keys()))
['a', 'c', 'b']
>>> # Updating does NOT change the order, though.
... od['a'] = 8
>>> print(list(od.keys()))
['a', 'c', 'b']
>>> # reversed() plays nice with OrderedDict.
... backwards = reversed(od.items())
>>> type(backwards)
<class 'odict_iterator'>
>>> print(list(backwards))
[('b', 7), ('c', 11), ('a', 8)]
```

Additional Methods

popitem(True)

Remove (key, value) pair, LIFO

popitem(False)

Remove (key, value) pair, FIFO

move_to_end(key, True)

Move existing key to end.

move_to_end(key, False)

Move existing key to beginning.

Beware!

`OrderedDict` lets you instantiate with keyword arguments. But it leaves the ordering completely undefined.

```
>>> bad = OrderedDict(a=2, b=3, c=7)
>>> list(bad.keys())
['b', 'a', 'c']
>>> # The .update() method works fine if you pass in an OrderedDict...
... new_ordered = OrderedDict([('d',9), ('e',6), ('f',32)])
>>> bad.update(new_ordered)
>>> list(bad.keys())
['b', 'a', 'c', 'd', 'e', 'f']
>>> # ... but doesn't define any ordering if you pass a regular dict.
... new_unordered = {'x': 14, 'y': 2, 'z': 23}
>>> bad.update(new_unordered)
>>> list(bad.keys())
['b', 'a', 'c', 'd', 'e', 'f', 'z', 'y', 'x']
```

More Collections

deque	Double-ended queue. Like list, with fast append/pop on each end
ChainMap (py3)	Single dict-like view of multiple mappings
Counter	for counting hashable objects
UserDict, UserList, UserString	Easier subclassing of built-ins

Details in the standard library:
https://docs.python.org/3/library/collections.html

Python 2 differences

- `ChainMap` only exists in Python 3.
- Python 2 puts `UserDict`, `UserList`, and `UserString` each in their own module.
- Some useful methods (like OrderedDict.move_to_end) only exist in Python 3.